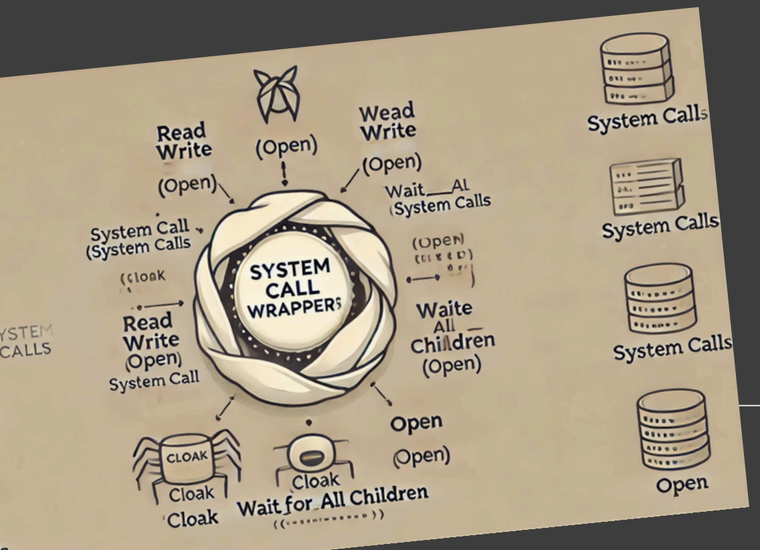
SysCall Wrappers

CS3500 Course Project

Group 3

Dev Mehta (CS22B007)
Kritang Kothari (CS22B012)
Aditya Jain (CS22B065)
Aditya Srivastava (CS22B066)
Raadhes Ch. (CS22B069)
Daksh Sehra (CS22B071)
Harshvardhan Daga (CS22B075)
Rohan Bagati (CS22B082)
Shreyanshu Gurjar (CS22B084)
Yashvardhan Toshniwal (CS22B088)
Mith Jain (CS22B097)



Presentation

Wrapping Syscalls Using Hooks

Aditya Srivastava, Dev Mehta

Hooks intercept system calls, redirecting them to custom functions, allowing modification or extension of behavior.

User-Space Hooking

- Modifies system calls like open(), read(), or write() by intercepting them at the application level, before or after they are executed.
- Create a wrapper function (e.g., logging the filename before calling the original open).
- Use dlsym to call the original syscall inside the wrapper.
- Use LD_PRELOAD to load the custom library.

Considerations

- Thread Safety: Make sure the wrapping does not interfere with other parts of the program, especially when multiple threads are involved.
- Security: Ensure that intercepting system calls does not introduce vulnerabilities or unintended behaviors.
- Performance: Adding extra steps before or after system calls can slow down execution, so efficiency is important.

Kernel-Level Hooking

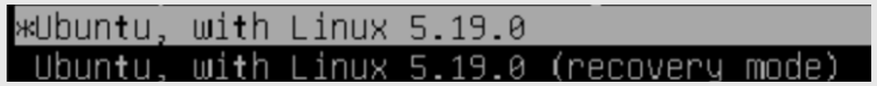
- Modifies system calls at the kernel level, which is the core of the operating system.
- Patch syscall table directly (requires kernel module).
- Replace syscall entry with a custom handler.
- Call original syscall from wrapper function.

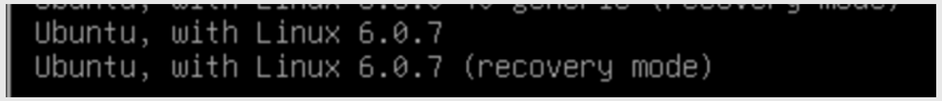
```c
// Example Fork Hooking
pid_t fork(void) {
    static pid_t (*original_fork)(void) = NULL;
    if (original_fork == NULL) {
        original_fork = dlsym(RTLD_NEXT, "fork");
        if (!original_fork) {
            fprintf(stderr, "Error in dlsym: %s\n", dlerror());
            return -1;
        }
    }
    // Do whatever
    // Call the real fork
    pid_t pid = original_fork();
    // Do whatever
    return pid;
}
```

Kernel Modifications

Yashvardhan, Raadhes

- Compilation:- Versions 4.17.4, 5.19 and 6.07 compiled. Linux 5.19 and 6.0.7 booted from grub.





- Modifications:- Kernel datastructures, syscall table, syscall definitions
  - Disabled custom System Call in special processes (sys_hello)
  - Resource Management system calls for forking
    - disable_fork():- Disable custom fork syscall for Current process and Descendants
    - fork_if_not_disable():- Custom fork syscall wrapper (kernel space)

C code

```c
int disable_fork(){
        return syscall(550);
}

int fork_if_not_disable(){
        return syscall(549);
}
```

syscalls.h

```
asmlinkage long sys_hello(void);
asmlinkage long sys_fork_if_not_disable(void);
asmlinkage long sys_disable_fork(void);
```

```
root@ubunt1:/home/rad/Documents# ./a.out
First custom fork call: 3462
Custom fork call after disable: -1
First custom fork call: 0
Custom fork call after disable: -1
```

  - close_all_files():- Custom system call to close all open files. Makes use of kernel helper functions.

Kernel Space Wrapper Implementation

- We tried implementing some wrappers in kernel space, compiled with the kernel code.
- Calling system calls from within kernel mode in Linux is generally not recommended because system calls are designed as user-space entry points into the kernel.
- To achieve the same functionality we have to use Kernel APIs like flip_open, vfs_read, wfs_write, kmalloc, printk/dmesg etc.
- Here is an example of a simple logger made using kernel APIs, which logs time of operation with a message taken as argument.
- The syscalls my_open and my_read call their functions which in turn calls the logger along with other functions to check permissions, validity of args etc.
- To define a custom syscall, use SYSCALL_DEFINE<N> (name , <args>) ; where N is the number of arguments your syscall requires.
- Then we need to add our new syscall to the syscall table (arch/x86/entry/syscalls/syscall_64.tbl) and syscalls definitions.

```c
void log_message(const char *message) {
    log_fp = filp_open(LOG_FILE, O_WRONLY | O_CREAT | O_APPEND, 0644);
    if (IS_ERR(log_fp)) {
        printk(KERN_ERR "Failed to open log file\n");
        return;
    }

    ktime_get_real_ts64(&ts);
    time64_to_tm(ts.tv_sec, 0, &tm);

    log_buf_size = snprintf(NULL, 0, "[%04ld-%02d-%02d %02d:%02d:%02d] %s\n",
                            tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
                            tm.tm_hour, tm.tm_min, tm.tm_sec, message) + 1;
    log_buf = kmalloc(log_buf_size, GFP_KERNEL);
    if (!log_buf) {
        printk(KERN_ERR "Failed to allocate memory for log buffer\n");
        filp_close(log_fp, NULL);
        return;
    }

    snprintf(log_buf, log_buf_size, "[%04ld-%02d-%02d %02d:%02d:%02d] %s\n",
             tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
             tm.tm_hour, tm.tm_min, tm.tm_sec, message);

    old_fs = get_fs();
    set_fs(KERNEL_DS);
    vfs_write(log_fp, log_buf, log_buf_size - 1, &pos);
    set_fs(old_fs);

    kfree(log_buf);
    filp_close(log_fp, NULL);
}
```

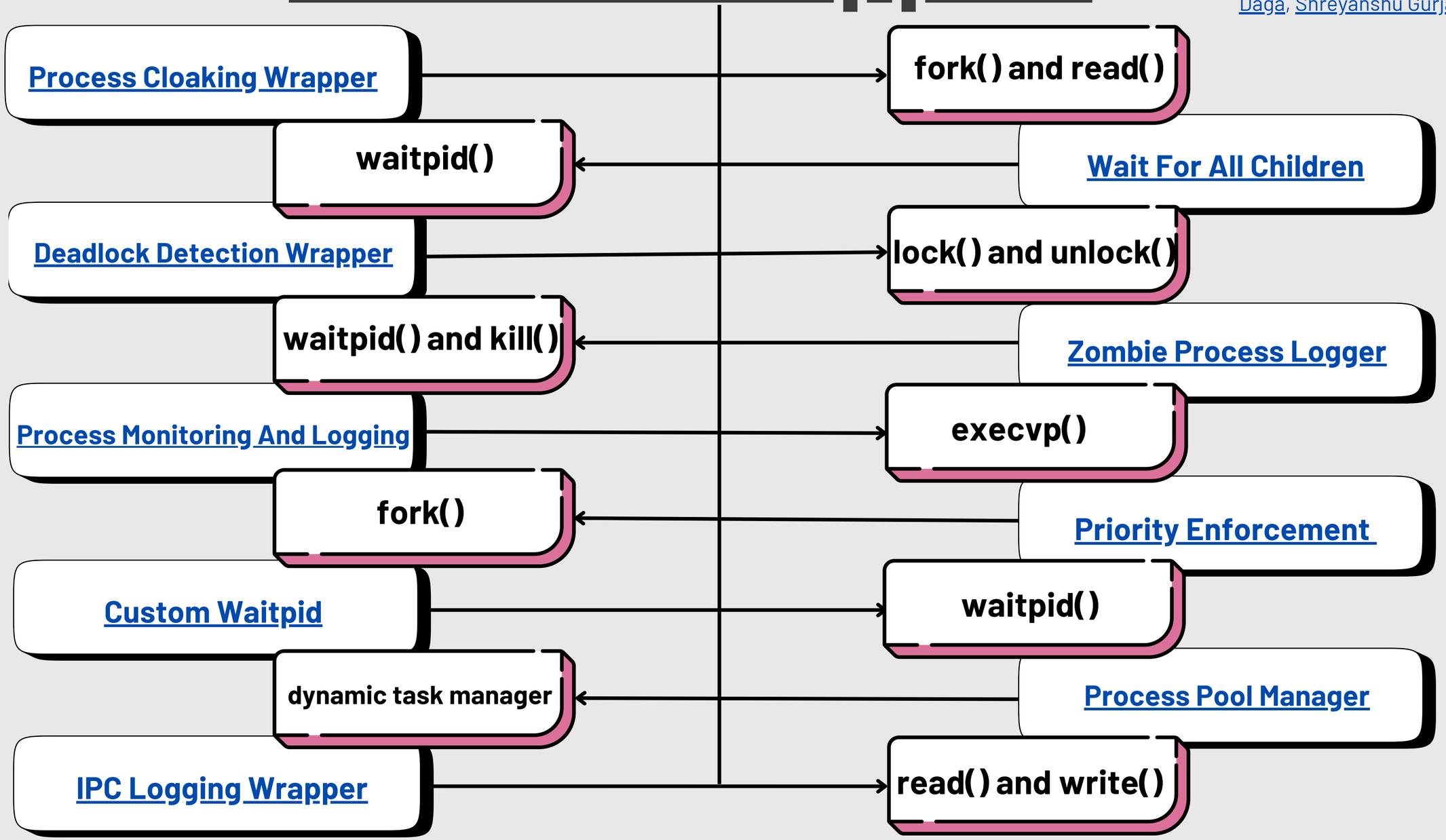
```c
SYSCALL_DEFINE3(my_open, const char*, pathname, int, flags, mode_t, mode) {
    return my_open(pathname, flags, mode);
}

SYSCALL_DEFINE3(my_read, int, fd, void*, buf, size_t, count) {
    return my_read(fd, buf, count);
}
```

syscall_64.tbl

```
547     x32     pwritev2                compat_sys_pwritev64v2
548     common  my_open                 sys_my_open
549     common  my_read                 sys_my_read
```

Process Wrappers

Aditya Jain, Aditya Srivastava, Harsh Vardhan Daga, Shreyanshu Gurjar



Process Wrappers

Aditya Jain, Aditya Srivastava, Harsh Vardhan Daga, Shreyanshu Gurjar

Wait For All Children

Hooked on waitpid.

The wrapper calls waitpid() on all the child processes of the calling process present in /proc/<PID>/task/<TID>/children directory.

Prevent Zombie Processes: Ensures no child process is left in a zombie state.

Deadlock Detection Wrapper

Wrapper intercepts system calls such as **mutex lock**, **unlock**, and **sem wait** and **post** to detect circular wait conditions between process and threads.

By tracking resource allocations and process dependencies, it detects **deadlocks** and terminates the process. It logs comprehensive details about the deadlock chain responsible.

Zombie Process Logger

This wrapper assumes that zombie processes may be created with **kill** or **waitpid** syscalls, and it proactively cleans up any zombies when these syscalls are called again.

Process Tracking: It monitors and logs process status, providing useful details on process termination and signal handling.

Process Pool Manager

An **attempt** to divide tasks among processes like threads but dynamically.

Each worker (child process) shares a pipe with the parent process, with the help of which, the parent process allots tasks to each child process to complete.

Process Cloaking Wrapper

The process cloaking wrapper intercepts the **fork** and **read** system calls to hide child processes from system monitoring tools like **ps or top**.

It captures the process from fork and **filters** it out from the output during **read calls**. This prevents the cloaked process from appearing in standard process listings, effectively "**hiding**" it from casual inspection.

Custom Waitpid

The wrapper adds a **timeout** to waitpid, killing the child process if it doesn't terminate in time, while avoiding blocking.

Non-Blocking: By using **WNOHANG** and a brief sleep interval, the wrapper ensures that the parent process doesn't block while waiting for the child process.

Possible extensions/improvements

Wait For all Children

The wrapper can further be extended to make the parent process wait for all the processes in it's process group to exit rather than just it's child processes.

Deadlock Detection Wrapper:

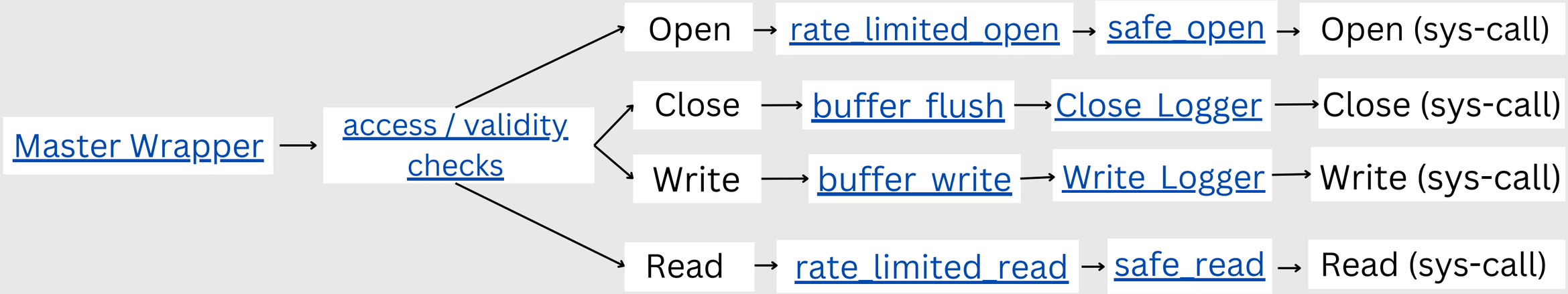
The wrapper can be extended further to support other threading and synchronization primitives such as condition variables.

Refer to the report for more details and the remaining wrappers
Refer to the report and codes for implementation details

File I/O Wrappers

Yashvardhan, Kritang, Daksh

- Master Wrapper
  - The file defines custom implementations of syscalls open(), read(), write(), close() using hooks to intercept them and run additional functions to enhance their behavior.
  - We have integrated all the previously made wrappers together. (see control flow graph below)
  - Each of the wrapper functions can be used separately as well.
  - Ex: Whenever a open syscall is made, it first checks access/control permissions, then checks the rate-limiter, and then goes to safe_open and logger to be opened and logged.
- Usage
  - Run make all to create the shared object file and compile main.c (with user code), or make filewrp.so to just create the shared object file.
  - You can execute the compiled user code with make run , or LD_PRELOAD=./filewrp.so ./<exec_name>
  - The log file is created in /tmp directory as file_io.log , it can be changed in the logger.h file.



File I/O Wrappers Customizations

Yashvardhan, Kritang, Daksh

- Every wrapper is easily customizable as it is a user level library.
- You can define custom logic and set appropriate metrics for access control, rate limiter, etc. , allowing them to be easily adapted or extended for different implementations.
- Error messages can be customized for every wrapper, making it easier to debug issues.

Wrapper-specific customisations

- *rate_limite*r
  - Our rate_limiter keeps a single counter for all the operations of a proces. This can be modified to keep different counters for different file descriptors.
  - The limit that we have defined for operations by a process can be changed by changing the values in rate_limiter.h (we have the huge time-span and low limits for demonstration purposes).
- safe_open
  - The warnings and errors that we have given are for the frequently encountered errors. The wrapper program can be easily modified with the help of different flags that are used in open() system call.
- control_permissions
  - The wrapper can be modified to check for a specific group or other criteria depending on the access control policy and file system implementation.

Buffer and Cache Wrappers for write()

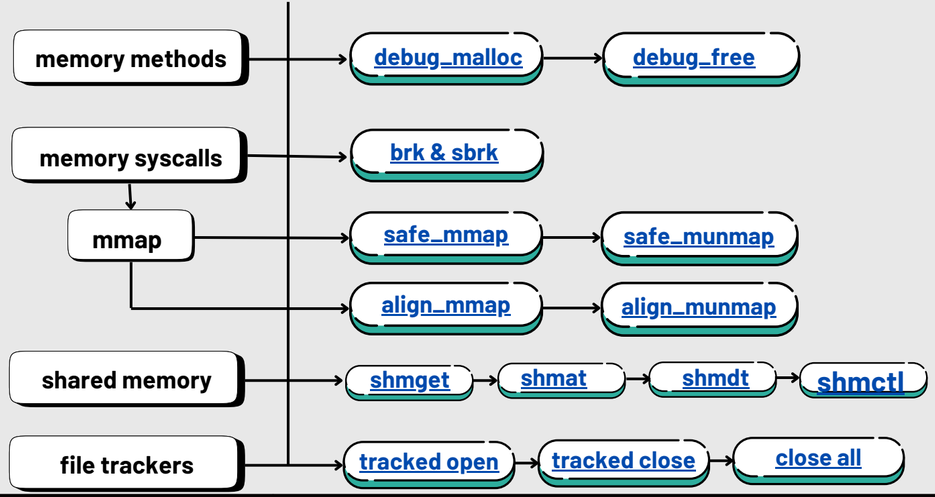
- We made a buffer wrapper for writes, but to make it more efficient it should be faster than normal file writes and access. Hence it should be in the main memory, now if multiple files are open we need to maintain different buffers for each and hence will need some of mapping to know which buffer is for which file descriptor. Using a hashtable is a very good way to implement both buffers and cache wrappers, but doing so in C is a bit difficult.

Memory Wrappers

Dev, Mith, Raadhes, Rohan

Refer to the report for more details and the remaining wrappers

Refer to the report and codes for implementation details



brk debug wrapper

Hooked on brk syscall

Used to dynamically allocate small memory chunks to a process, for malloc etc.
Logging of brk system calls for debugging

Ensuring thread safety in cases where glibc ptmalloc2 is not used

mmap wrappers

Hooked on mmap syscall

safe_mmap - Logs requested size, flags, resulting pointer, error messages if mmap fails, verifies valid pointers and adds to tracking.

aligned_mmap - Reserves a memory region larger than required size, computes aligned address using bitwise operations.

munmap wrappers

Hooked on munmap syscall

safe_munmap - Validates pointers, logs freed memory size, address, errors and updates tracking structures to remove freed memory.

aligned_munmap - Retrieves the original base address stored by aligned_mmap and calculates the total allocated size.

debug malloc and free

Tracks and logs allocations with error detection and safeguards.
Tracking: Uses a linked list (AllocRecord) for pointers and sizes.
Errors: Logs failed or zero-size allocations.
Safety: Detects double-free/invalid pointers, ensures thread safety with locks or TLS

shared memory

Hooked on respective shm syscalls

Wrappers for the commonly used shared memory functions like shmget, shmat, shmdt, shctl.

memory pool wrapper

Custom user wrapper that manages memory spaces in terms of pools of data, for obtaining large regions of memory at once.

Helpful for programs that frequently access small chunks of memory, and improved reference of locality.

Memory Wrappers

Dev Mehta, Mith Jain

Extensions/Customizations

Memory Access Pattern Analysis

- Identifying inefficient memory access patterns can help optimize programs, especially in terms of cache utilization.
- Track the frequency of memory accesses (e.g., how often a particular memory region is read/written). This can help identify hotspots in the program and suggest areas for optimization

Heap Corruption Detection

- Heap corruption can occur if there are invalid memory accesses or buffer overflows.
- Check for heap corruption after each memory allocation or deallocation. This can be done by performing integrity checks on the heap structure. One approach by comparing checksums has been implemented in the project.

Garbage Collection

- All these wrappers can be efficiently used for custom garbage collection. Since the tracking structure anyway stores the access information, many garbage collection algorithms can be realised in the code.
- A basic example using generic mark and sweep algorithm has been implemented in the project, which stores links between memory locations and marks reachable locations, and hence sweeping the unmarked ones.